EECS 481 — Software Engineering
Winter 2019 — Exam #2

- **Write your UM uniqname and UMID and your name on the exam.**
- There are thirteen (13) pages in this exam (including this one) and six (6) questions, each with multiple parts. Some questions span multiple pages. If you get stuck on a question, move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two page-sides of notes.
- Even vaguely looking at a cellphone or similar device (e.g., tablet computer) during this exam **is cheating**.
- Please write your answers in the space provided on the exam. Clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We may deduct points if your solution is far more complicated than necessary.
  - *Good Writing Example:* Testing is an expensive activity associated with software maintenance.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that small portion (rounded down) for not wasting time.**

UM uniqname: ANSWER KEY

UM ID: ANSWER KEY

Name (print): ANSWER KEY

UM uniqname: (yes, again!) ANSWER KEY

UM ID: (yes, again!) ANSWER KEY

Problem	Max points	Points
1 — Delta Debugging	20	
2 — Requirements	18	
3 — Design	21	
4 — Productivity	18	
5 — Other Topics	23	
Extra Credit	0	
TOTAL	100	

How do you think you did? ______________________________

1 Delta Debugging (20 points)

Consider applying the Delta Debugging algorithm to the task of selecting off-the-shelf libraries and components to add to a software project. A finite set of capabilities $C = \{c_1, \ldots, c_n\}$ exists; a subset $D \subseteq C$ is necessary to complete the project. A finite set of third-party libraries $L = \{l_1, \ldots, l_m\}$ is available. Each library l_i has an associated set of positive and/or negative capabilities given by a function $\mathsf{caps} : L \to \mathcal{P}(C \times \mathcal{B})$, where $\mathcal{B}$ denotes the Booleans.

For example, it could be that $\mathsf{caps}(l_3) = \{(\text{graphics}, \text{true}), (\text{iOS}, \text{false})\}$, meaning that l_3 is a library that provides "graphics" but does not work with "iOS". Similarly, if $\mathsf{caps}(l_7) = \{(\text{authentication}, \text{true}), (\text{iOS}, \text{true})\}$, then l_7 is a library that provides "authentication" support and does run on "iOS". However, we could *not* use l_3 and l_7 at the same time because one uses iOS and the other forbids it: they *mismatch* on that capability.

We want to find a small set of libraries possible such that (1) all of the required capabilities are satisfied, and (2) there are no mismatches on any capabilities. Formally, we want to find an answer $A \subseteq L$ such that (1) $\forall c_i \in D.\ \exists l_j \in A.\ (c_i, \text{true}) \in \mathsf{caps}(l_j)$, and (2) $\forall c_i \in C.\ \forall l_j \in A.\ (c_i, \text{true}) \in \text{caps}(l_j) \Rightarrow \neg \exists l_k \in A.\ (c_i, \text{false}) \in \text{caps}(l_k)$. (That logical formulation says the same thing as the English; you can skip it if you like. There are no "tricks" in the math.) A set of libraries is *interesting* if all of the required capabilities are positively satisfied with no mismatches anywhere.

(2 pts.) In general, this problem formulation violates at least one of the fundamental assumptions of the basic Delta Debugging algorithm. Identify one such unmet assumption.

This formulation is not *monotonic* and is not *unambiguous*. Either is full credit.

(10 pts.) Provide a simple example that shows that your chosen assumption is violated. You must do so with $n \leq 3$ and $m \leq 3$ by giving definitions for C, D, L and caps.

Not monotonic: because of conflicts, adding more libraries can make things worse. Consider $C = \{\text{graphics}, \text{iOS}\}$ and $D = \{\text{graphics}\}$. Suppose there are two libraries and $\mathsf{caps}(l_1) = \{(\text{graphics}, \text{true}), (\text{iOS}, \text{false})\}$ while $\mathsf{caps}(l_2) = \{(\text{iOS}, \text{true})\}$. We observe that $\{l_1\}$ alone is interesting (it satisfies D) but $\{l_1, l_2\}$ together are not interesting (they have a conflict).

Not unambiguous: Some may view this is "cheap", but as phrased, nothing in this formulation prevents you from having two possible disjoint interesting solutions. Consider $C = \{\text{graphics}\}$ and $D = \{\text{graphics}\}$. Suppose there are two libraries and $\mathsf{caps}(l_1) = \mathsf{caps}(l_2) = \{(\text{graphics}, \text{true})\}$. Note that $\{l_1\}$ and $\{l_2\}$ alone are both interesting: they both satisfy the single requirement. But the intersection of those two sets is the empty set, which is not interesting, so unambiguity is violated.

(4 pts.) Consider the following claim: "using Delta Debugging to minimize failure-inducing code changes is unnecessary if your development process includes continuous integration testing." In no more than four sentences, support or refute this claim.

Likely "refute" — continuous integration testing typically only runs short tests (e.g., unit tests). While some projects run every test on every commit, it remains quite common to have more expensive tests (e.g., end-to-end integration tests) run one a nice or over the weekend. In such a setting we could have a few patches that all pass the unit tests individually, but that, combined, fail the nightly integration test. DD would help figure out the relevant subset. The key aspect here is that even if you use CIT for some tests, you may still have other tests that are not part of CIT and that would thus benefit from DD.

(4 pts.) Our formulation of Delta Debugging included a check to $\mathsf{Interesting}(P \cup P_1)$ and $\mathsf{Interesting}(P \cup P_2)$. Explain why both checks will never return true at the same time. (You should walk through the reasoning or otherwise explain the contradiction that occurs if both are true; a "one word" answer will not suffice.)

We obtained P_1 and P_2 by *partitioning* the input set C. That means that P_1 and P_2 are disjoint: $P_1 \cap P_2 = \emptyset$. They do not share any elements. In some calls to DD (such as the first one), P can be empty.

By the Unambiguous assumption, if X and Y are both interesting then $X \cap Y$ must also be interesting. In this case, since $\mathsf{Interesting}(P_1)$ and $\mathsf{Interesting}(P_2)$, we would have $\mathsf{Interesting}(P_1 \cap P_2)$ which means $\mathsf{Interesting}(\emptyset)$, which violates DD's input setup (if the empty set is interesting, then by montonicity everything is interesting, and the problem is trivial).

2 Requirements (18 points)

(4 pts.) List an *informal goal* and a *verifiable requirement* for the same quality (i.e., non-functional) property. Why is the first "merely" informal? How would you verify the second? (A four-sentence answer should suffice.)

A possible answer was given in the slides for "ease of use":

Informal goal: "the system should be easy to use by experienced controllers, and should be organized such that user errors are minimized." This is informal because words like "easy to use", "experienced" and "minimized" are not quantified and can't be tested.

Verifiable non-functional requirement: "Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day, on average." This is verifiable because you could run that experiment (i.e., give someone the training and count the number of errors per day afterwords) and see if the requirement was met or not.

(6 pts.) Consider the following claim: "Branch coverage metrics simply approximate the notion of traceability from requirements elicitation." Support or refute the claim, giving very brief definitions for the key terms. (You should not need more than six sentences.)

Likely "support" — Statement coverage is based on the notion that we have no confidence that non-covered lines are bug-free. For covered lines we gain some confidence (but it's not a proof). Branch coverage extends this by checking conditionals (and thus the data values they depend on). But the implicit assumption is still "we don't know which lines and branches matter, so we better make sure we check all of them". If we had traceability to requirements (e.g., "passing Test T1 means you have met requirements R1 and R2, passing test T2 means you have met requirement R3, etc.") then the coverage would be basically irrelevant. If you know you meet all of the requirements, you don't need to worry about coverage (presumably uncovered lines are dead code or never happen in real life somesuch). You almost never have full traceability, but if you did, it would supplant coverage.

(8 pts.) Consider the following descriptions adapted from a Mutation Testing Homework Assignment. For each one, write "**F**" if it best describes a functional requirement, "**Q**" if it best describes a quality (non-functional) requirement, and "**N**" if it does not describe a requirement. (If you think there is some overlap, pick the *best* or most precise answer.)

_F_ You must implement and support three mutation operators: negating comparisons, swapping binary operators, and deleting statements.

_Q_ Your submission must run to completion on the autograder within two minutes.

_F_ A correct implementation will distinguish test suite A from test suite B.

_N_ Mutants that violate Python's indentation rules will not efficiently help mutation analysis. This is a true statement, but you don't "have to" do it. It's not a requirement. You can create a few inefficient "dead" mutants and still get full credit.

_Q_ Your submission must not attempt to reverse engineer or in any way communicate or reveal information about the held-out autograder tests. This is a privacy or confidentiality requirement. Imagine that instead of talking about held-out autograder tests, this was talking about hospital patient medical records or credit card numbers.

_F_ Your solution must be deterministic. If your solution is non-deterministic and gets the wrong answer some of the time, it is violating a functional property.

_F_ The mutants must be named 0.py, 1.py, 2.py, and so on.

_N_ The ast.parse, ast.NodeVisitor, ast.NodeTransformer, and astor.to_source portions of the interface are relevant for this assignment. This is a true statement, but you don't "have to" use (all) of them. It's just very likely. Even if everyone in class coincidentally does X, that's not the same as X being a requirement.

3 Design (21 points)

Consider the following *incorrect* code for implementing an *Observer* (i.e., Publish-Subscribe) design pattern, adapted from James Perretta's lecture.

```
1  class Subject {
2    public static void subscribe (Observer observer) {
3      subscribers.Add (observer);
4    }
5    public static void unsubscribe (Observer observer) {
6      subscribers.Remove (observer);
7    }
8    public static void change_state () {
9      foreach (Observer observer in subscribers) {
10       observer.update ();
11       subscribers.Remove (observer);
12     }
13   }
14   private static List < Observer > subscribers = new List < Observer > ();
15 }
```

(5 pts.) In at most three sentences, indicate the bug in this code and how you would fix it.

The bug is that `subscribers.Remove (observer);` on line 11 is extraneous. In the Observer pattern you do not remove subscribers after each update; instead, they remain and receive all updates until they unsubscribe. You fix the bug by deleting that extraneous line.

(5 pts.) Consider the Template Method Pattern and the notion of Design by Contract. How do the use of virtual methods, non-virtual methods, and access modifiers (public, private, etc.) help enforce the pre- and post-conditions of the algorithm being implemented using this design pattern?

The Template Method pattern: "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure."

The "template method" defined in the base class is non-virtual. Since this base class non-virtual method calls the derived class protected virtual method and not the other way around, we have some guarantee that the code before and after the virtual method calls will happen every time, thus helping to enforce the pre- and post-conditions of the algorithm being implemented.

Note that the base class method can even explicitly enforce pre- and post-conditions, e.g. by doing error checking on input parameters and the return value.

To put it another way, Template Method derived classes can only change the parts of the algorithm that the base class allows them to.

Some credit may be awarded for discussing how the base class can use access modifiers to control derived class access to member variables and methods.

It is true that when overriding a method, the subclass can have *weaker* pre-conditions and can provide *stronger* post-conditions if desired (but not the other way around). However, this is not what makes the template method pattern "stand out" here. A fully correct answer should discuss the fact that the base class "template method" itself is non-virtual, which allows it to enforce pre- and post-conditions without the derived classes messing them up.

(5 pts.) Consider the following claim: "A tool that symbolically generates human-readable What-style documentation (e.g., for exceptional conditions or code changes) is actually a tool that generates test oracles." Support or refute the claim. (You should not need more than four sentences.)

Likely "refute" — tools that symbolically generate human-readable What-style documentation use the same constraint-based approach used by tools that generate test inputs (e.g., AFL). We described this approach in class: it involves gathering up all of the branch conditions. A test input generator solves those conditions to get an input; a documentation generator prints them out in English to get documentation. However, they are all based on what the code *currently does*, rather than what the code *should do*. You could make an argument that they generate test *inputs* (not oracles), but there is no reason to believe that documentation synthesized from the current code is correct (what if the current code is buggy?).

However, you could do "support" — it would be harder, but you could consider the use case where you *know* (by other means) that the code is currently correct. Then you generate documentation for it. You could then save that documentation and use it as a test oracle to prevent future *regressions*.

(6 pts.) Consider the following claim: "Since reading code (e.g., in a code review or code inspection) takes time per line and since code comprehension is the dominant activity in software maintenance, the most effective design strategy for software maintenance is to write correct programs that contain fewer lines of code." Support or refute this claim. (You should not need more than five sentences. You may assume the reader is familiar with all references from the lectures or readings.)

Likely "refute" — from the code review and inspection slides, we know that the recommended reading rate is about 400 LOC per hour (Slide 54) and that people get tired after an hour (Slide 53), so it is true that if you have a smaller program, it takes less time to read the whole program. In addition, we saw in the Productivity lecture that the amount of code you can write per day, over the course of the entire project, is a small constant (Slide 24). However, trying to make the program as small as possible is almost certainly a *perverse incentive*, as per the Measurement lecture (Slide 55). For example, we know from the Code Inspection lecture that beacons and descriptive variable names really help (Slide 29), but descriptive variables and comments take up space. Consider this analogy:

(1) What we really want is code that is easy to maintain. We observe that it takes less time to read smaller code. Therefore, we should make code small at all costs. (This is bad logic.)

(2) What we really want is code that is easy to read. We observe that it is easier to read code that has more newlines. Therefore, we should add a million newlines at all costs. (This is

bad logic.)

This problem relates to “perverse incentives”.

4 Productivity (18 points)

(6 pts.) Consider the following quotation adapted from Laurie Williams et al.'s *Strengthening the Case for Pair Programming*:

> Programmer-hours do not double with pair programming as one might expect. First, collaboration improves the problem-solving process. As a result, far less time is spent in the chaotic, time-consuming compile and test phases . . . Additionally, pairs find that by pooling their joint knowledge, they can conquer almost any technical task immediately. In teaching the university class that participated in the experiment, the instructor noticed that individual workers asked two or three times more technical questions than did collaborative workers. While waiting for the answers to these questions, these students were generally unproductive . . . Pair programmers put pressure on each other to perform, keeping their partner focused and on task.

How do you reconcile this claim with Fred Brooks' law "adding human resources to a late software project makes it later" and suggestion that software engineering is not a partitionable activity? (Convince us that both claims can be true at once. This may involve bringing in knowledge of pair programming or the Mythical Man Month not mentioned here.)

This is a difficult, qualitative argument. But, like many issues in requirements elicitation, it is likely an issue of confusing vocabulary. We can approach it by looking at each of the key words (e.g., "late", "partitionable activity")

"Partitionable" – This is probably the best argument to go after. Brooks argues that you can't really divide up writing the code (etc.), but pair programming is explicitly not proposing to do that. The driver "actively types at the computer or records a design" while the a navigator "watches the work of the driver and attentively identifies problems, asks clarifying questions, and makes suggestions." They are *explicitly doing different things*: pair programming is not proposing to partition "literal coding by having two humans type at the same keyboard at the same time" (which would likely fail) but instead "high level software development". While one is blocked (see Productivity slides) the other can help out.

"Adding" — Brooks argues that adding more people to a late project makes it later. However, in pair programming, you have the second person present from the beginning. The second person sees the code as it is being written (indeed, the second person helps brainstorm how the code should be written). Part of the Brooks argument was "it takes time for new people to get up to speed on a project, so if you bring some new people in, you'll lose a few days just teaching them about the system". Here you are paying that cost from the start: the second person already knows the lay of the land.

"Late" — A final reconciliation would be that Brooks argues that adding more people makes it slower, and pair programming has been found to make things 15–100% slower, so there's actually no conflict if the slowdown is 100%. (This is a little cheap: Brooks would predict that adding a second person would make things twice as late, so the question is "why is the overhead for pair programming less than 2x"?)

The last two arguments are unlikely to be full credit alone. Student answers vary significantly.

You are responsible for *giving* a non-behavioral technical interview to job candidates; you are the interviewer. Your company views technical interviews as an assessment of software engineering skills. The programming problem you ask of candidates is:

> Write MoveNode(), a variant on Push(). Instead of creating a new node and pushing it onto the given list, MoveNode() takes two lists, removes the front node from the second list and pushes it onto the front of the first.

The candidate's complete response is below. The first two commented lines indicate questions the candidate asked you.

```c
/* Q: Can the source list be empty? A: No. */
/* Q: Can the lists overlap? A: Yes. */

void MoveNode(struct node** destRef, struct node** sourceRef) {
  struct node* newNode = *sourceRef; // the front source node
  assert(newNode != NULL);
  *sourceRef = newNode->next; // Advance the source pointer
  newNode->next = *destRef; // Link the old dest off the new node
  *destRef = newNode; // Move dest to point to the new node
}

// test: {1,2,3} {1,2,3} -> {1,1,2,3} {2,3}
```

(4 pts.) Identify two thing that the candidate did well.

The code is functionally correct. Descriptive comments are present. The identifier names are clear. The indenting style is consistent.

(8 pts.) Identify and justify four significant things that the candidate did poorly.

No "why" documentation is given. No attempt was made to elicit quality requirements. No indication is made of the running time. Only one test case is present, and it is very poor: no "negative" or "corner case" tests are present. No discussion is made of design, maintainability or similar concerns. No evidence is given that the code is correct if the lists overlap.

5 Other Topics (23 points)

(4 pts.) List and briefly explain two difficulties that may arise when attempting to use branch coverage in a multi-language project.

1. Most tools only support one language, so coverage information from two sources may have to be combined manually for reporting.
2. It is not clear whether you should include branch coverage of the "glue code".
3. Instrumentation for branch coverage may conflict with multi-language glue code and native interfaces. (Instrumentation wants to rewrite your code to record branches, but the native interface code may be too fragile to be rewritten in that way.)
4. Tools that automatically generate test data to maximize branch coverage may not work on multi-language projects (AFL is a rare counter-example).

(4 pts.) List two things that change in the brain as humans gain expertise. Your answers can be associated with general expertise and/or computing expertise.

1. The shape of the brain physically changes, as evidenced in the London taxi cab driver study where the hippocampus grew with more taxi experience.
2. The efficiency of the brain changes, resulting in a lower metabolic activity required to carry out the same task.
3. The brains of experts treat programming language tasks more like natural language tasks.

(4 pts.) Kate Highnam's guest lecture mentioned the possibility of changing managers multiple times while working on the same project. Choose "planning", "testing" or "requirements". Explain two aspects of your choice that would be complicated by changing your direct manager during a project.

Varies. An interesting angle would be to consider "requirements" with the manager as a stakeholder. Different managers often give different priorities to various requirements: as you move from manager M1 to manager M2, you may need to communicate again to resolve conflicts. "Planning" is also a likely concern: different managers will have different biases and approaches to measurement, for example, and will thus have different estimates for project completion. This was a fairly open-ended question.

(5 pts.) Consider the following computer science tasks:

1. Find a number in an unsorted array.
2. Find a number in a sorted array.
3. Find a number in a balanced binary search tree.
4. Find a number in an unbalanced tree.
5. Reverse an unsorted array.

In one sentence, explain how experts and novices cluster problems. Then indicate how novices would likely divide the problems into two clusters (give the problem numbers the novice would put in the first partition, then the problem numbers the novice would put in the second). Finally, indicate how expert CS theoreticians would likely divide them.

Novices cluster tasks based on gross physical features (such as "does this have a pulley" or "does this have an array") while experts cluster tasks based on the underlying theory or law of physics (such as "Newton's Second Law" or "Big-Oh notation").

So we expect novices to cluster (1,2,5) vs. (3,4) — all of the array problems vs. all of the tree problems.

We expect "CS theoretician" experts to cluster (1,4,5) vs. (2,3) — the first three all take $\mathcal{O}(N)$ time and visit every element, while the last two take $\mathcal{O}(\log N)$ time and only visit part of the structure. Experts were still limited to just two clusters.

(1 pt. each) Read the following narrative adapted from quotes from the Reetu Das guest lecture. Fill in each ____ blank with the single *most specific or appropriate* corresponding concept from the answer bank. (Each ____ blank does have a corresponding answer.) Each option from the answer bank will be used *at most once*.

A. Code Inspection	B. Debugger	C. Design Pattern	D. Fault Localization
E. Functional Property	F. Native Interface	G. Quality Assurance	H. Quality Property
I. Signal Coverage	J. Stakeholder	K. Technical Interview	L. Triage

__G__ We finished our design in May of last year but we are still verifying it. The verification step here is part of QA. This is almost "waterfall": do all writing before all testing.

__B__ Verifying hardware was difficult because it had less visibility: we did not use GDB, just signals. Fault Loc is tempting, but not as specific.

__J__ You start with one person, but that person isn't the one you want to talk to. So you keep following pointers until you get to the right person. This is describing identifying the right stakeholders to talk to.

__E__ Doctors won't use your stuff unless every bit of the output matches the expected value. Even if our output accuracy is actually better, they won't be happy.

__H__ One important aspect of this domain is security. We must anonymize everything; you can't share patient data on the cloud. Security and privacy are functional properties.

__F__ We transfer data between C and Verilog. For Reetu, C was the base language an Verilog was the low-level hardware, but the idea is the same.

6 Extra Credit **(1 pt each; we are tough on reading questions)**

(Feedback) In retrospect, what is one thing you enjoyed about the second half of this class? What should be changed about the second half of this class for next year?
(Free/Participation) If you could give one piece of advice to yourself when you were starting this class, what would it be?
(Your Choice Reading 1) Identify one of the readings from the second half of the class that we did not cover explicitly during the lecture. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here — sorry!).
(My Choice Reading 1) In "The Economic Value of Rapid Response Time", characterize the reported relationship between system response time and the number of transactions a user could complete in an hour.

There is a nonlinear relationship: adding X delay to the system for Y transactions causes a total slowdown or *more* than XY. This was used by managers and others to argue that expensive employees need faster computers.

(My Choice Reading 2) In "Industrial Experience with Design Patterns", name one of the three claims or lessons that all six companies attributed to design patterns.

"good communications medium", "extracted from working designs", and "capture design essentials" were common to all six companies.

(My Choice Reading 3) On the forum, Microsoft Program Manager Peter Shultz gave an industry perspective on the class material. Explain the "persuasive" relationship between Program Managers and Software Engineering Managers he detailed.

At least at Microsoft, PMs don't actually tell software engineers what to do. Rather, program managers persuade software engineering managers** that certain things need to get done. If our arguments are persuasive, software engineering managers will allocate their software engineers to work on what we think matters most. Software engineering managers often call this "funding" a certain piece of work (i.e. "we'll fund that feature you seem to care about so much").